Tornado 4.3 官方文档中文翻译

ZEY LEE

Published
with GitBook

目錄

Tronado简介	0
用户指南	1
简介	1.1
异步非阻塞	1.2
协程	1.3
web框架	2
tornado.web — RequestHandler 和 Application	2.1
tornado.template — 灵活的输出	2.2
tornado.escape — 转义和字符串操作	2.3
tornado.locale — 国际化支持	2.4
tornado.websockets — 浏览器双向通信	2.5
HTTP服务器和客户端	3
tornado.httpserver — 非阻塞http服务器	3.1
tornado.httpclient — 异步http客户端	3.2
tornado.httputil — 操作http headers和urls	3.3
tornado.http1connection — HTTP/1.x 客户端/服务器 的实现	3.4
异步网络	4
tornado.ioloop — 主事件循环	4.1
tornado.iostream — 非阻塞socket的便利封装	4.2
tornado.netutil — 各种网络工具	4.3
tornado.tcpclient — IOStream 工厂	4.4
tornado.tcpserver — 基于TCP服务器的IOStream	4.5
协程和并发	5
tornado.gen — 简化异步代码	5.1
tornado.concurrent — threads和futures并行工作	5.2
tornado.locks — 同步事物	5.3
tornado.queues — 协程队列	5.4
tornado.process — 多进程工具	5.5
其他服务	6
tornado.auth — 使用OpenID和OAuth的第三方登录	6.1

tornado.wsgi — 与python其他框架和服务互通 6.2
tornado.platform.asyncio — asyncio和Tornado的桥梁 6.3
tornado.platform.caresresolver — 使用C-Ares的异步DNS解决方案 6.4
tornado.platform.twisted — Twisted和Tornado的桥梁 6.5
实用工具 7
tornado.autoreload — 在开发中自动检测代码改动 7.1
tornado.log — 日志支持 7.2
tornado.options — 命令行工具 7.3
tornado.stack_context — 异步回调中的异常操作 7.4
tornado.testing — 异步代码的单元测试工具 7.5
tornado.util — 常用工具 7.6

Tornado 4.3 中文翻译

Tornado是一个基于python实现的web框架和异步网络库，最初用来开发FriendFeed.通过使用非阻塞的网络I/O模型，Tornado可以抗住上千的并发连接，所以在长轮询、websockets以及那些彼此长连接的应用来说变得很简单

快捷链接

- Tornado 4.3 版本下载：
- 源码
- 邮件组：
- stackovreflow
- wiki

Hello world

一个使用tornado实现的『Hello World』程序

```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

def make_app():
    return tornado.web.Application([
        (r"/", MainHandler),
    ])

if __name__ == "__main__":
    app = make_app()
    app.listen(8888)
    tornado.ioloop.IOLoop.current().start()
```

这个例子没有用到任何tornado异步特性

Tornado安装

自动安装

`pip install tornado`

Tornado被PyPI收录，所以可以直接使用pip或者easy_install安装。注意，使用PyPI或者easy_install安装的tornado包括一些未被实现的demo应用，所以你最好再下载一个tar的源码为好.

手动安装

Download tornado-4.3.tar.gz:

```
tar xvzf tornado-4.3.tar.gz
cd tornado-4.3
python setup.py build
sudo python setup.py install
The Tornado source code is hosted on GitHub.
```

使用须知: Tornado 4.3 运行在python 2.6 2.7 或者3.2等更好版本（对python2.6和3.2的支持将会在下一个发行版中移除）。对于python2，tornaod改进了对SSL的支持，强烈推荐2.7.9或更新版本。

用户指南

- 简介
- 异步非阻塞I/O
  - 简介
  - 异步非阻塞
- web框架
  - tornado.web

简介

Tornado是一个基于python实现的web框架和异步网络库，最初用来开发FriendFeed.通过使用非阻塞的网络I/O模型，Tornado可以抗住上千的并发连接，所以在长连接、websockets以及那些彼此长连接的应用来说变得很简单

Tornado can be roughly divided into four major components:

Tornado可以粗略地分成4个主要组件： 一个网路框架（包括RequestHandler，它是所有web应用和其他辅助类的父类） 基于HTTP实现的客户端和服务端（HTTPServer和AsyncHTTPClient） 一个异步的网络库（IOLoop 和 IOStream），它控制HTTP网络模块，同时也可以被用来实现其他协议 一个协程库（tornado.gen），相对于直接使用回调函数，它提供了一个更直白的方式去写异步代码

Tornado网络框架和HTTP服务器一起组成一个WSGI的全栈替代品。单独在WSGI容器中使用tornado网络框架或者tornaod http服务器，有一定的局限性，为了最大化的利用tornado的性能，推荐同时使用tornaod的网络框架和HTTP服务器

异步非组赛I/O

实时网络需要一个可以长时间保持的连接和最小cpu消耗。对于传统同步网络服务器，这意味着每个用户都要去分配一个线程，代价是非常大的。

为了追求并发连接的最小成本，Tornaod使用一个单线程的事件循环，这意味着所有的应用都必须是异步且非阻塞，一个时间点只有一个操作被执行。

异步和非阻塞有很强的关联性，经常被混淆，但请记住，他们是两回事儿

阻塞

当一个函数在返回之前等待某些资源时，这个函数就会阻塞。 一个函数阻塞的原因可能有多重：网络I/O，磁盘I/O，锁等等。实际上，每一个函数在运行使用cpu时发生阻塞，会占用一点内存。举一个极端的例子，为什么CPU阻塞比其他阻塞来的严重得多？想想bcrypt的密码hashing函数，在CPU上用了很长的时间，相对于网络和磁盘访问来说。

一个函数在某些时候会阻塞，当然也有不阻塞的时候。例如，在DNS解析的时候，tornado.httpclient使用默认配置的话就会阻塞，但是在访问其他网络时不会阻塞（为了缓解这个情况，使用ThreadedResolver或者基于适当配置的libcurl的tornado.curl_httpclien）。文章里面我们谈论的阻塞都是说的网络I/O，尽管所有的的阻塞都被最小化了。

异步 一个异步函数在它完成之前就会返回，通常在应用触发某些动作之前会产生一些后台进程。异步接口有这些风格： 回调参数 返回一个占位符（） 交给队列 回调函数注册

不管使用上面哪一种类型，异步函数对于调用者来说都有不同的含义。想要实现对于调用者来说完全透明的一个同步函数实现异步功能，并非易事。（gevent系统使用轻量级的线程，性能跟异步系统不相上下，但它并没有使用异步）

举例 一个同步函数

```
from tornado.httpclient import HTTPClient

def synchronous_fetch(url):
    http_client = HTTPClient()
    response = http_client.fetch(url)
    return response.body
```

同一个函数，使用回调参数的异步实现：

```
from tornado.httpclient import AsyncHTTPClient

def asynchronous_fetch(url, callback):
    http_client = AsyncHTTPClient()
    def handle_response(response):
        callback(response.body)
    http_client.fetch(url, callback=handle_response)
```

同一个函数，使用回调函数的实现：

```
from tornado.concurrent import Future

def async_fetch_future(url):
    http_client = AsyncHTTPClient()
    my_future = Future()
    fetch_future = http_client.fetch(url)
    fetch_future.add_done_callback(
        lambda f: my_future.set_result(f.result()))
    return my_future
```

Future老版本非常复杂，但仍然推荐在tornado中使用它，因为它有两个主要优点。错误处理在Future.result中高度一致，只需要抛出一个异常，此外Future很容易和协程配合。Coroutines将会在后面的章节讨论。这里是一个coroutine实现的样例程序，和之前同步的版本非常相似：

```
from tornado import gen

@gen.coroutine
def fetch_coroutine(url):
    http_client = AsyncHTTPClient()
    response = yield http_client.fetch(url)
    raise gen.Return(response.body)
```

The statement raise gen.Return(response.body) is an artifact of Python 2 (and 3.2), in which generators aren’t allowed to return values. To overcome this, Tornado coroutines raise a special kind of exception called a Return. The coroutine catches this exception and treats it like a returned value. In Python 3.3 and later, a return response.body achieves the same result.

`raise gen.Return(response.body)` 在python2和3.2中是人为搞出来的，生成器是不允许有返回值的。 为了避免这个，coroutines抛出一个名为 `Return` 的异常。coroutines捕获这个异常，把它当做一个返回值来处理。 在Python3.3或更新版本中， `return response.body` 也实现了这个。

Coroutines

协程

官方推荐使用协程的方式来完成异步的代码。协程使用python的 `yield` 关键字来挂起和继续，代替原来的一系列回调（像gevent有时候会把轻量级的线程当做是协程来用，但是在tornado里，所有的协程都有明确的上下文环境，被当做是异步函数）

Coroutines are almost as simple as synchronous code, but without the expense of a thread. They also make concurrency easier to reason about by reducing the number of places where a context switch can happen. 协程几乎像同步一样简单，几乎没有线程开销。

Example:

from tornado import gen

@gen.coroutine def fetch_coroutine(url): http_client = AsyncHTTPClient() response = yield http_client.fetch(url)

```
# In Python versions prior to 3.3, returning a value from
# a generator is not allowed and you must use
#   raise gen.Return(response.body)
# instead.
return response.body
```

Python 3.5: async and await Python 3.5 introduces the async and await keywords (functions using these keywords are also called “native coroutines”). Starting in Tornado 4.3, you can use them in place of yield-based coroutines. Simply use async def foo() in place of a function definition with the @gen.coroutine decorator, and await in place of yield. The rest of this document still uses the yield style for compatibility with older versions of Python, but async and await will run faster when they are available:

async def fetch_coroutine(url): http_client = AsyncHTTPClient() response = await http_client.fetch(url) return response.body The await keyword is less versatile than the yield keyword. For example, in a yield-based coroutine you can yield a list of Futures, while in a native coroutine you must wrap the list in tornado.gen.multi. You can also use tornado.gen.convert_yielded to convert anything that would work with yield into a form that will work with await.

While native coroutines are not visibly tied to a particular framework (i.e. they do not use a decorator like tornado.gen.coroutine or asyncio.coroutine), not all coroutines are compatible with each other. There is a coroutine runner which is selected by the first coroutine to be called, and then shared by all coroutines which are called directly with await. The Tornado

coroutine runner is designed to be versatile and accept awaitable objects from any framework; other coroutine runners may be more limited (for example, the asyncio coroutine runner does not accept coroutines from other frameworks). For this reason, it is recommended to use the Tornado coroutine runner for any application which combines multiple frameworks. To call a coroutine using the Tornado runner from within a coroutine that is already using the asyncio runner, use the tornado.platform.asyncio.to_asyncio_future adapter.

How it works A function containing yield is a generator. All generators are asynchronous; when called they return a generator object instead of running to completion. The @gen.coroutine decorator communicates with the generator via the yield expressions, and with the coroutine's caller by returning a Future.

Here is a simplified version of the coroutine decorator's inner loop:

Simplified inner loop of tornado.gen.Runner

def run(self):

```
# send(x) makes the current yield return x.
# It returns when the next yield is reached
future = self.gen.send(self.next)
def callback(f):
    self.next = f.result()
    self.run()
future.add_done_callback(callback)
```

The decorator receives a Future from the generator, waits (without blocking) for that Future to complete, then "unwraps" the Future and sends the result back into the generator as the result of the yield expression. Most asynchronous code never touches the Future class directly except to immediately pass the Future returned by an asynchronous function to a yield expression.

How to call a coroutine Coroutines do not raise exceptions in the normal way: any exception they raise will be trapped in the Future until it is yielded. This means it is important to call coroutines in the right way, or you may have errors that go unnoticed:

@gen.coroutine def divide(x, y): return x / y

def bad_call():

```
# This should raise a ZeroDivisionError, but it won't because
# the coroutine is called incorrectly.
divide(1, 0)
```

In nearly all cases, any function that calls a coroutine must be a coroutine itself, and use the yield keyword in the call. When you are overriding a method defined in a superclass, consult the documentation to see if coroutines are allowed (the documentation should say that the method “may be a coroutine” or “may return a Future”):

@gen.coroutine def good_call():

```
# yield will unwrap the Future returned by divide() and raise
# the exception.
yield divide(1, 0)
```

Sometimes you may want to “fire and forget” a coroutine without waiting for its result. In this case it is recommended to use IOLoop.spawn_callback, which makes the IOLoop responsible for the call. If it fails, the IOLoop will log a stack trace:

The IOLoop will catch the exception and print a stack trace in

the logs. Note that this doesn't look like a normal call, since

we pass the function object to be called by the IOLoop.

IOLoop.current().spawn_callback(divide, 1, 0) Finally, at the top level of a program, if the `.IOLoop` is not yet running, you can start the IOLoop, run the coroutine, and then stop the IOLoop with the IOLoop.run_sync method. This is often used to start the main function of a batch-oriented program:

run_sync() doesn't take arguments, so we

must wrap the

call in a lambda.

IOLoop.current().run_sync(lambda: divide(1, 0)) Coroutine patterns Interaction with callbacks To interact with asynchronous code that uses callbacks instead of Future, wrap the call in a Task. This will add the callback argument for you and return a Future which you can yield:

@gen.coroutine def call_task():

```
# Note that there are no parens on some_function.
# This will be translated by Task into
#   some_function(other_args, callback=callback)
yield gen.Task(some_function, other_args)
```

Calling blocking functions The simplest way to call a blocking function from a coroutine is to use a ThreadPoolExecutor, which returns Futures that are compatible with coroutines:

thread_pool = ThreadPoolExecutor(4)

@gen.coroutine def call_blocking(): yield thread_pool.submit(blocking_func, args) Parallelism The coroutine decorator recognizes lists and dicts whose values are Futures, and waits for all of those Futures in parallel:

@gen.coroutine def parallel_fetch(url1, url2): resp1, resp2 = yield [http_client.fetch(url1), http_client.fetch(url2)]

@gen.coroutine def parallel_fetch_many(urls): responses = yield [http_client.fetch(url) for url in urls]

```
# responses is a list of HTTPResponses in the same order
```

@gen.coroutine def parallel_fetch_dict(urls): responses = yield {url: http_client.fetch(url) for url in urls}

```
# responses is a dict {url: HTTPResponse}
```

Interleaving Sometimes it is useful to save a Future instead of yielding it immediately, so you can start another operation before waiting:

@gen.coroutine def get(self): fetch_future = self.fetch_next_chunk() while True: chunk = yield fetch_future if chunk is None: break self.write(chunk) fetch_future = self.fetch_next_chunk() yield self.flush() Looping Looping is tricky with coroutines since there is no way in Python to yield on every iteration of a for or while loop and capture the result of the yield. Instead, you'll need to separate the loop condition from accessing the results, as in this example from Motor:

import motor db = motor.MotorClient().test

@gen.coroutine def loop_example(collection): cursor = db.collection.find() while (yield cursor.fetch_next): doc = cursor.next_object() Running in the background PeriodicCallback is not normally used with coroutines. Instead, a coroutine can contain a while True: loop and use tornado.gen.sleep:

@gen.coroutine def minute_loop(): while True: yield do_something() yield gen.sleep(60)

Coroutines that loop forever are generally started with

spawn_callback().

IOLoop.current().spawn_callback(minute_loop) Sometimes a more complicated loop may be desirable. For example, the previous loop runs every 60+N seconds, where N is the running time of do_something(). To run exactly every 60 seconds, use the interleaving pattern from above:

@gen.coroutine def minute_loop2(): while True: nxt = gen.sleep(60) # Start the clock. yield do_something() # Run while the clock is ticking. yield nxt # Wait for the timer to run out. Next Previous

阿道夫

`tornado.web` — `RequestHandler` and `Application` classes

`tornado.web` 是一个具有异步特性的简单web框架，它能抗住大规模的连接，非常适合长轮询请求。

这是一个简单的『Hello, world』应用例子：

```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

if __name__ == "__main__":
    application = tornado.web.Application([
        (r"/", MainHandler),
    ])
    application.listen(8888)
    tornado.ioloop.IOLoop.current().start()
```

更多信息请查看用户指南.

线程安全

一般来说， `RequestHandler` 中的方法或者是Tronado中其他的方法都不是线程安全的。这里要注意，有些方法比如 `write()` ，`finish()` ，`flush()` 只能在主线程中被调用。如果你使用多线程，使用 `IOLoop.add_callback` 在请求结束之前把控制权交给主线程是非常重要的。

Request handlers

> `class tornado.web.RequestHandler(application, request, **kwargs)[source]`

HTTP请求handlers的基类 子类必须最少使用下面入口中的一种方法

入口

> `RequestHandler.initialize()[source]`

子类初始化的钩子 第三个参数必须使用传入一个带有关键字的字典给 `initialize()`

举例：

```
class ProfileHandler(RequestHandler):
    def initialize(self, database):
        self.database = database

    def get(self, username):
        ...

app = Application([
    (r'/user/(.*)', ProfileHandler, dict(database=database)),
    ])
```

> `RequestHandler.prepare()[source]`

在 `get/post/etc` 之前被调用

可以重写这个函数实现自定义的初始化，不管当前请求的方法是什么

Asynchronous support: Decorate this method with gen.coroutine or return_future to make it asynchronous (the asynchronous decorator cannot be used on prepare). If this method returns a Future execution will not proceed until the Future is done. 异步支持：装饰 `gen.coroutine` 或者